Window Programming
Visual C++ MFC Programming

Lecture 02

김예진
Dept. of Game Software

Notices

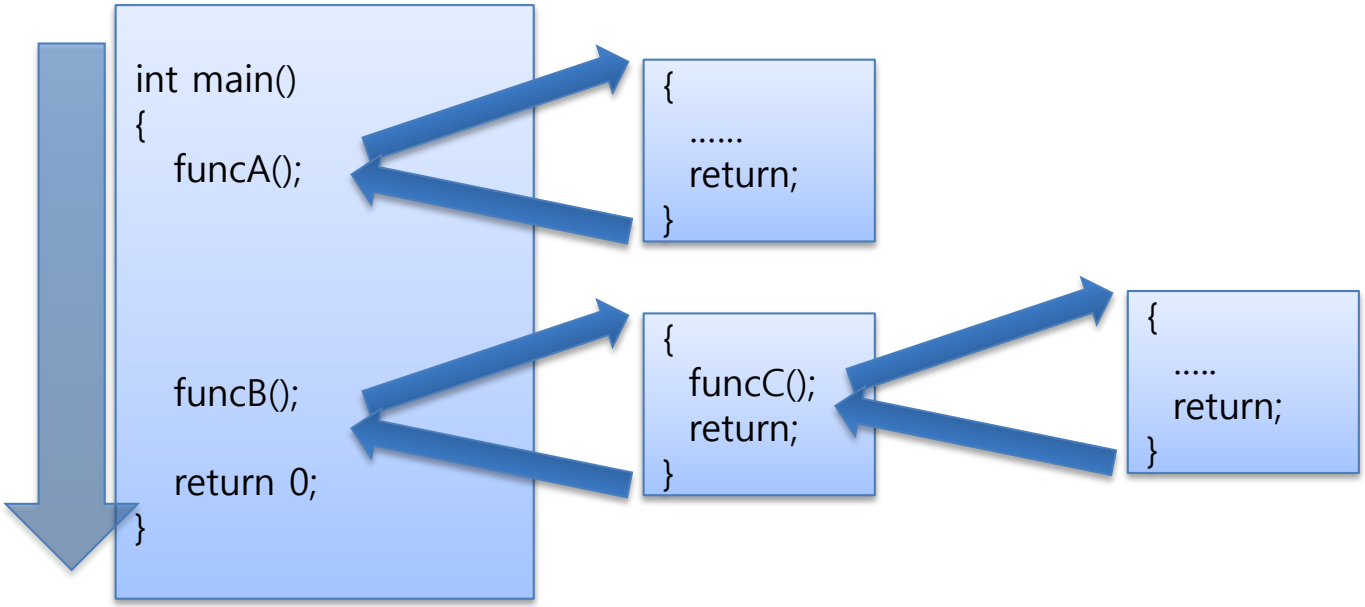
- 03/07: 502 → 501 등록 이동
- 03/21: HW 1 (Due: 03/28)

Plan

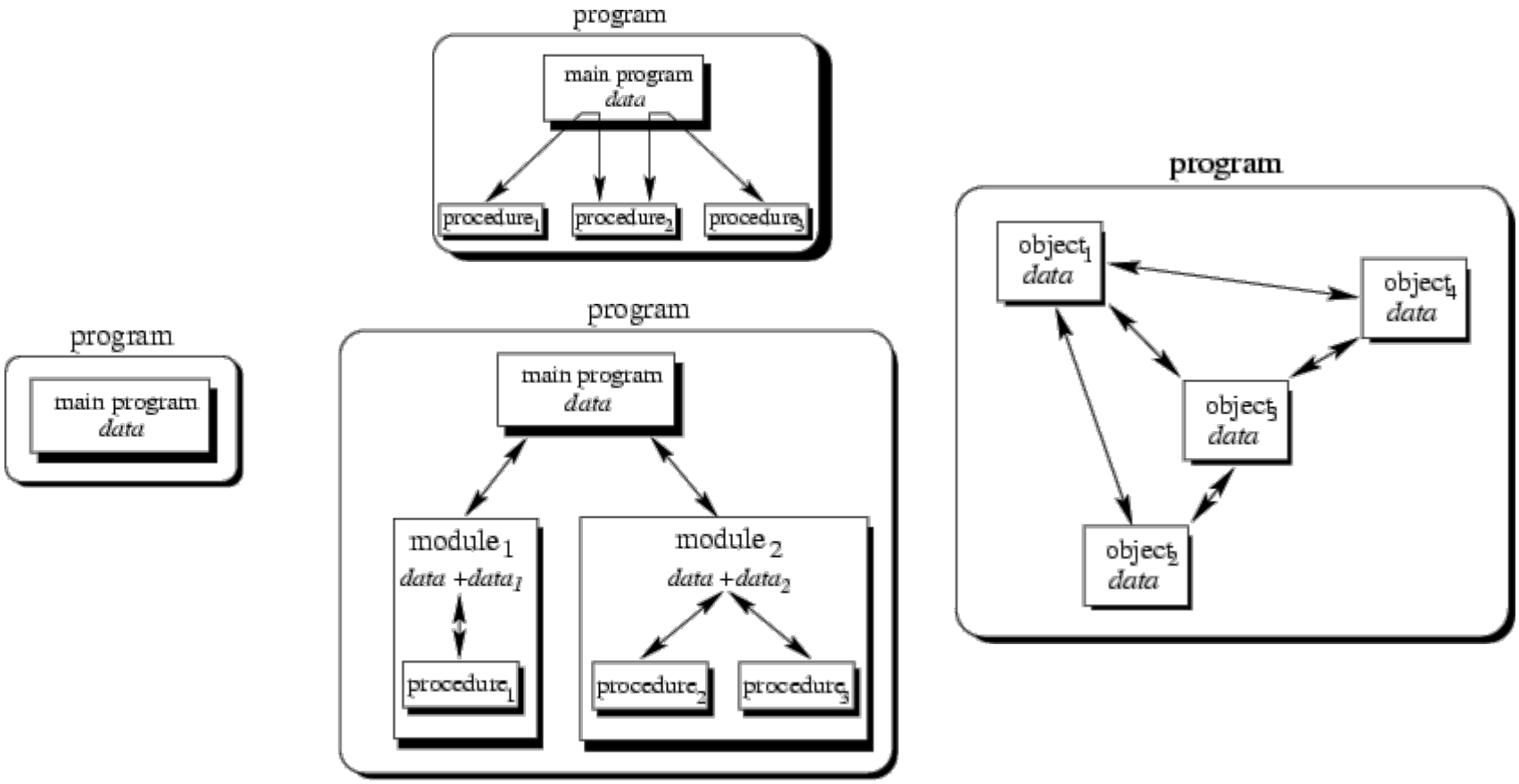
- 절차지향과 객체지향
- C++ 프로그래밍 기초
  - Class 구조체
  - Inheritance(상속)과 containment(포함)
  - Virtual function(가상 함수)와 polymorphism(다형성)

절차지향과 객체지향

- 절차지향 프로그래밍
  - main() 함수로 시작
  - main() 함수가 return하면 끝남
  - 다른 함수를 차례대로 호출함
  - 기본적으로 main() 함수가 다른 함수들을 (언제, 어디서) 호출하는 구조



절차지향과 객체지향



비구조적 구조적/절차지향 객체지향

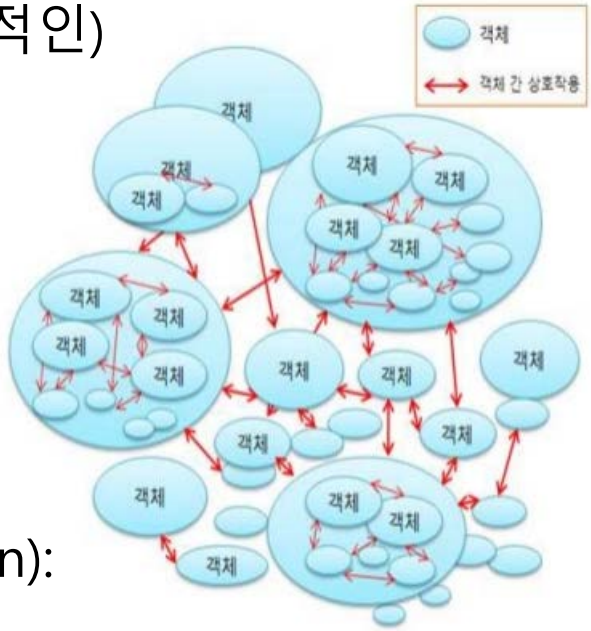
절차지향과 객체지향

- 객체지향의 배경
  - Software module의 재사용과 독립성을 강조

- 객체지향과 절차지향의 비교
  - 절차지향(procedural-oriented): data 구조와 그 data를 변화시키는 function으로 구성
  - 객체지향(object-oriented): 객체들이 **메시지(message)**를 통하여 통신함으로써 원하는 결과를 얻음. 각 객체는 고유의 **속성(attribute)**과 data를 처리할 수 있는 **메소드(method)**로 구성

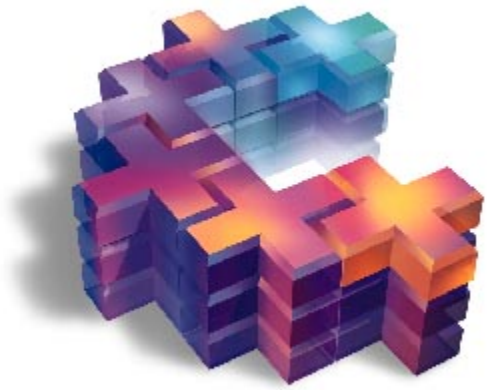
절차지향과 객체지향

- 객체(object)
  - 효율적으로 정보를 관리하기 위하여, 사람들이 의미를 부여하고 분류하는 논리적인(개념적인) 단위
  - 실 세계에 존재하는 하나의 단위에 대한 software적 표현
  - 관련된 **변수**와 **함수**의 묶음
- 객체의 구성
  - 속성의 값을 나타내는 data: **attribute**
  - Data를 변경하거나 조작하는 기능(function): **method**



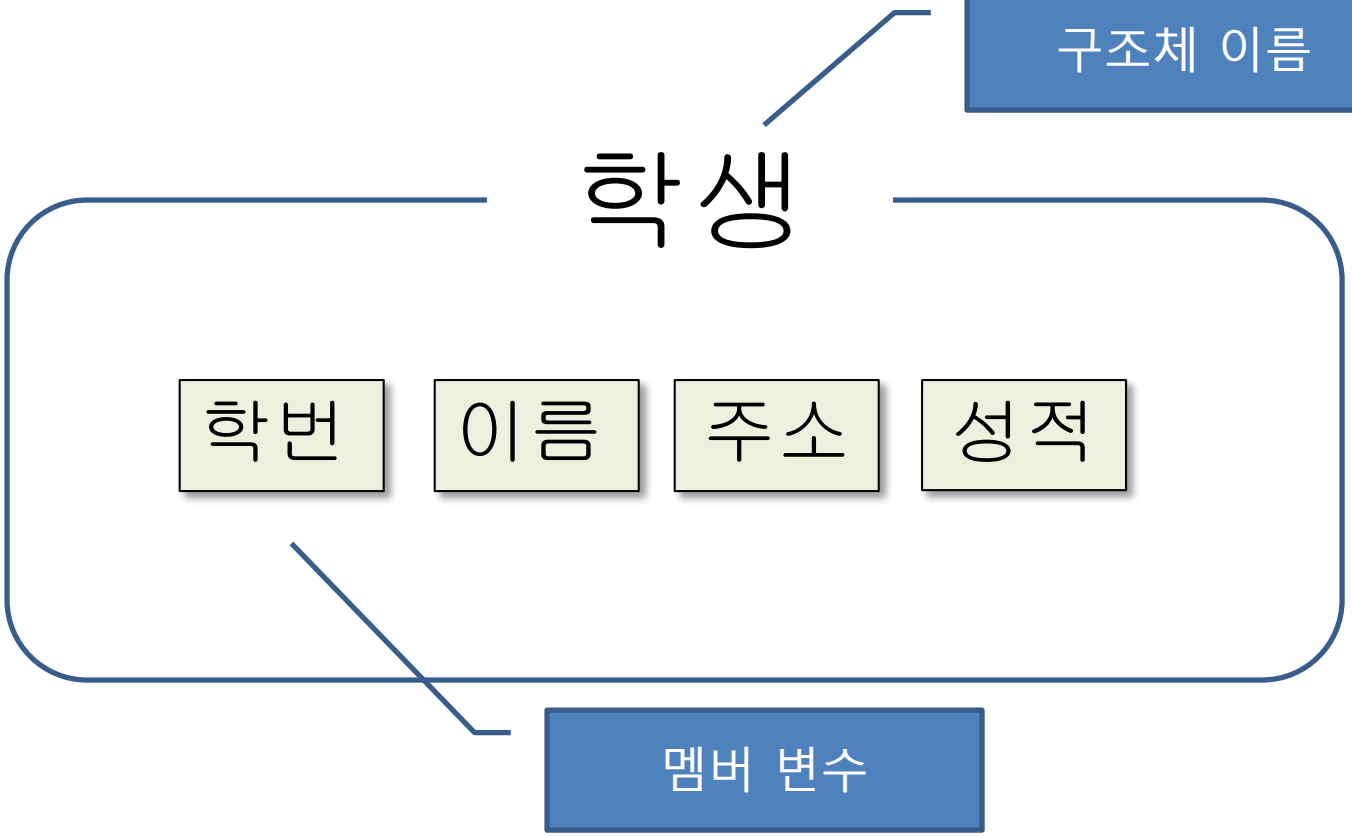
C++ 프로그래밍 기초

- C++
  - 향상된 C
  - C + Class 구조체
    - C의 효율성 + 객체지향(object-oriented) 개념



C++ 프로그래밍 기초

- 구조체(structure)
  - Data를 묶는 단위
  - 관련된 정보를 그룹화하여 표현



C++ 프로그래밍 기초

- 구조체의 정의: struct

```
struct 구조체이름
{
        멤버변수타입1 변수이름1;
        멤버변수타입2 변수이름2;
        멤버변수타입3 변수이름3;
        …
};
```

```
struct student
{
        int id;
        char name[30];
        std::string address;
        float grade;
};

void main()
{
        student a, b;
}
```

C++ 프로그래밍 기초

- 구조체 member로의 접근

구조체변수이름.맴버변수이름

```
struct student
{
	int id;
	char name[30];
	std::string address;
	float grade;
};

void main()
{
	student a, b;
	a.id = 1001, b.id = 1002;
	strcpy(a.name, "John"), strcpy(b.name, "Peter");
	a.address = "Sejong", b.address = "Seoul";
	a.grade = 3.0, b.grade = 3.5;
}
```

C++ 프로그래밍 기초

- 구조체를 가리키는 pointer
  - 변수형처럼 사용해서 pointer를 정의할 수 있음

```
struct rectangle
{
     int x, y;
     int width, height;
};

void main()
{
     rectangle rc;
     rectangle *p = &rc;
}
```

C++ 프로그래밍 기초

- 구조체 member의 변수 접근
  - 구조체에서는
  - 구조체 pointer는

구조체변수.맴버변수

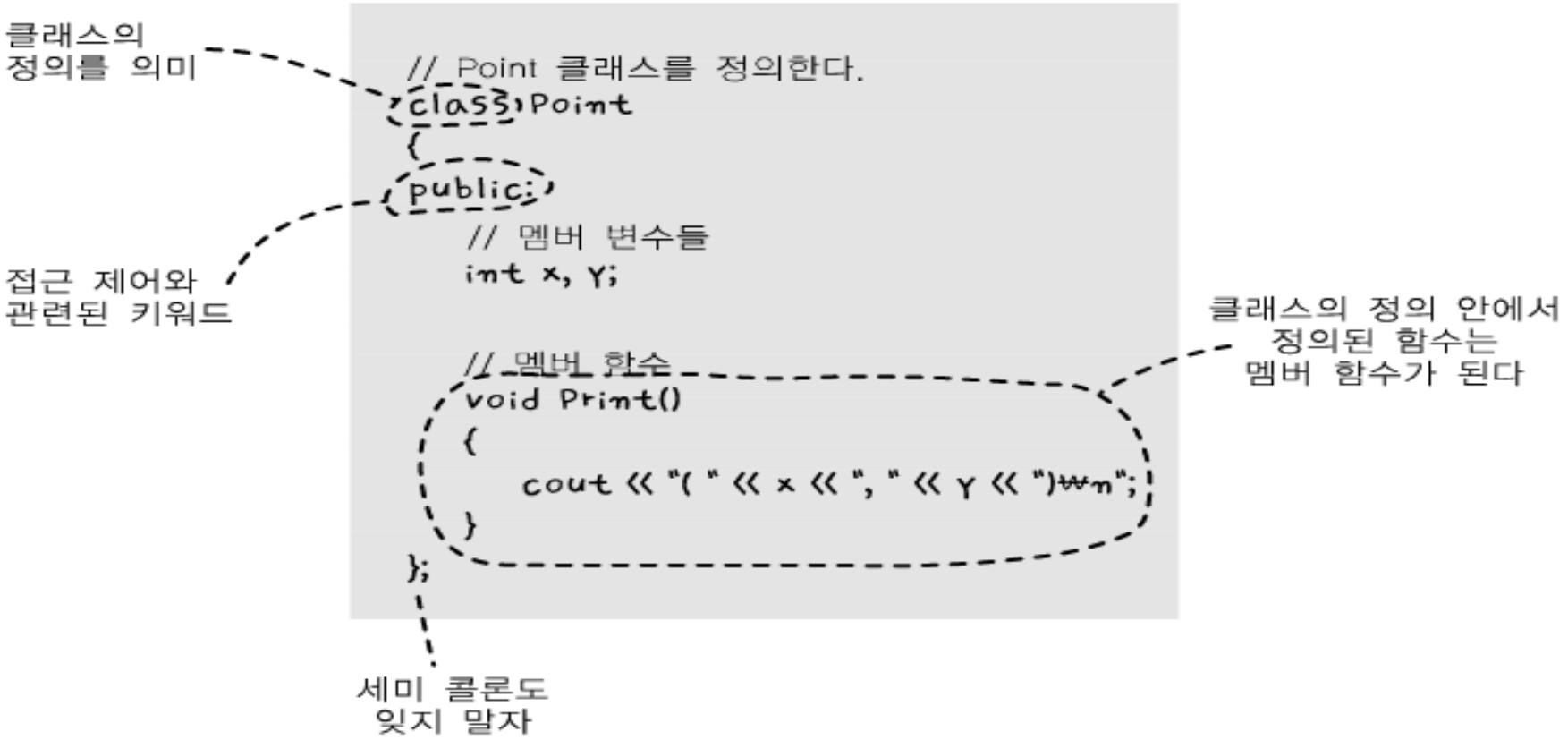
(*구조체포인터).맴버변수

구조체포인터->맴버변수

```
struct rectangle
{
     int x, y;
     int width, height;
};

void main()
{
     rectangle rc;
     rectangle *p = &rc;

     rc.x = 10;     //
     (*p).x = 10;  // 모두 같음
     p->x = 10;     //
}
```

C++ 프로그래밍 기초

- Class의 정의
  - 데이터와 그 기능을 묶는 단위



C++ 프로그래밍 기초

- Class의 정의
  - Point class를 정의하는 예

```
// main.cpp
#include <iostream>

using namespace std; // std::을 생략하기 위해 필요

// Point 클래스를 정의
class Point
{
// 접근권한 설정
public:
          // 멤버 변수들
          int x, y;

          // 멤버 함수
          void Print()
          {
                    cout << "( " << x << ", " << y << ")\n";
          }
};
```

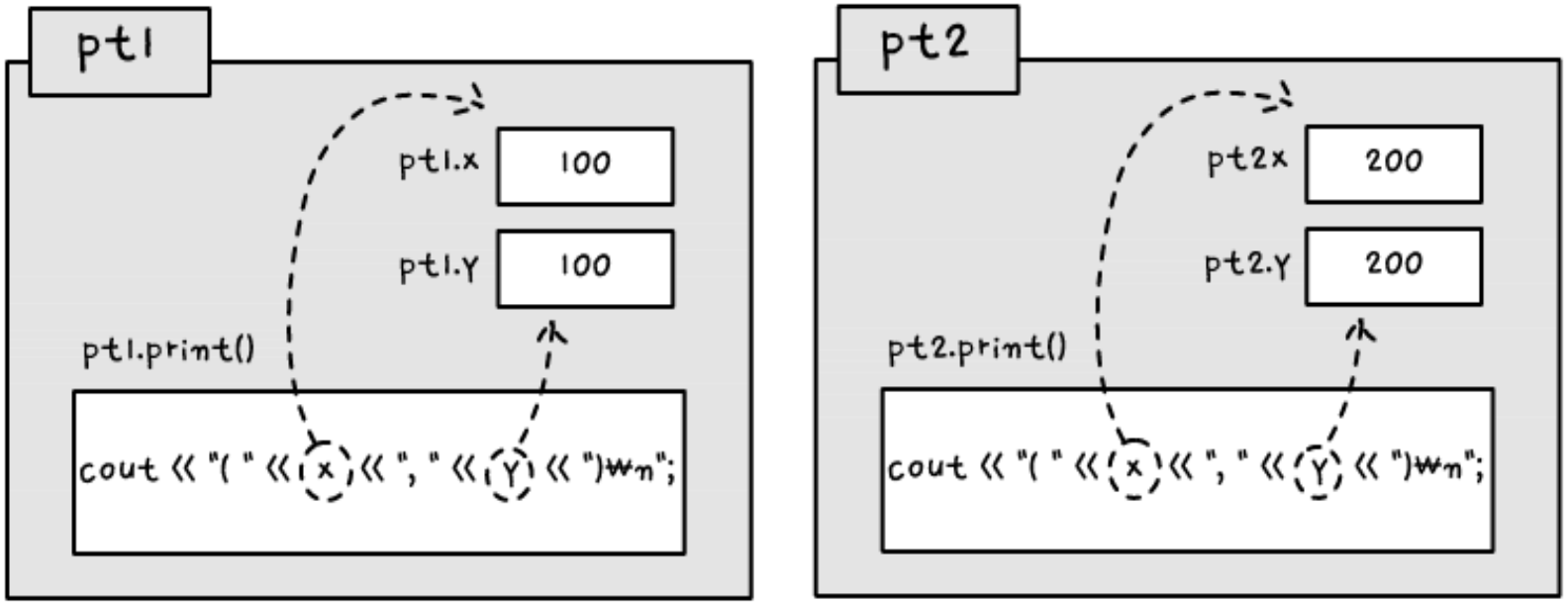
C++ 프로그래밍 기초

- 객체의 생성과 사용
  - Class 객체를 생성하고 사용한 예

```
// main.cpp
int main()
{
     // 객체를 생성
     Point pt1, pt2;

     // pt1, pt2를 초기화
     pt1.x = 100;
     pt1.y = 100;
     pt2.x = 200;
     pt2.y = 200;

     // pt1, p2의 내용을 출력
     pt1.Print();
     pt2.Print();

     return 0;
}
```

C++ 프로그래밍 기초

- 객체의 생성과 사용
  - Print() 함수에서 사용하는 x, y의 의미



C++ 프로그래밍 기초

- Member 함수의 위치
  - Class의 바깥쪽에 member 함수를 정의한 예

```
// main.cpp
class Point
{
public:
          // 멤버 변수
          int x, y;

          // 멤버 함수
          void Print();
};

void Point::Print()
{
          cout << "( " << x << ", " << y << ")\n";
}
```

C++ 프로그래밍 기초

- Class의 선언(declaration)과 정의(definition) 분리
  - Class의 선언: header(*.h 파일)
  - Class의 정의: source(*.cpp 파일)

```
// point.h: Point class의 member 함수 선언
class Point
{
public:
            int x, y;
            void Print();
};

// point.cpp: Point class의 member 함수 정의
#include "point.h"

void Point::Print()
{
            cout << "( " << x << ", " << y << ")\n";
}

// main.cpp
#include "point.h"

int main()
{
            ...
}
```

C++ 프로그래밍 기초

- Class의 생성자(constructor)와 소멸자(destructor)
  - 생성자는 객체를 생성할 때 자동으로 호출되는 함수
    - 그러므로 생성자는 객체를 사용할 수 있도록 초기화 하는 코드를 넣기에 알맞은 장소
  - 소멸자는 객체를 소멸할 때 자동으로 호출되는 함수
    - 그러므로 소멸자는 객체가 사용한 리소스를 정리하는 코드를 넣기에 알맞은 장소

C++ 프로그래밍 기초

- 기본 생성자(default constructor)
  - 기본 생성자를 추가한 예

```
// point.h
class Point
{
public:
            int x, y;
            void Print();
            Point();
};

// point.cpp
Point::Point()
{
            x = 0;
            y = 0;
}

// main.cpp
int main()
{
            Point pt;               // 생성자 호출
            pt.Print();
}
```

C++ 프로그래밍 기초

- 매개변수(parameter)가 있는 생성자
  - 여러 종류의 생성자 추가 예

```
// point.h
class Point
{
public:
            int x, y;
            void Print();
            Point();
            Point(int initX, int initY);
            Point(int initX2, int initY2); // O.k.?
            Point(int initX, float initY); // O.k.?
};

// point.cpp
Point::Point(int initX, int initY)
{
            x = initX;
            y = initY;
}

// main.cpp
int main()
{
            Point pt(3, 5);
            pt.Print();
}
```

function overloading

C++ 프로그래밍 기초

- 소멸자(destructor)
  - 소멸자를 사용해서 할당한 메모리를 해제한 예

```
class Point
{
public:
        int *arr;
        Point();
        ~Point();
};

Point::Point()
{
        // 동적 메모리 할당
        int arraySize = 10;
        arr = new int [arraySize];
}

Point::~Point()
{
        // 메모리 해제
        delete[] arr;
        arr = NULL;
}
```

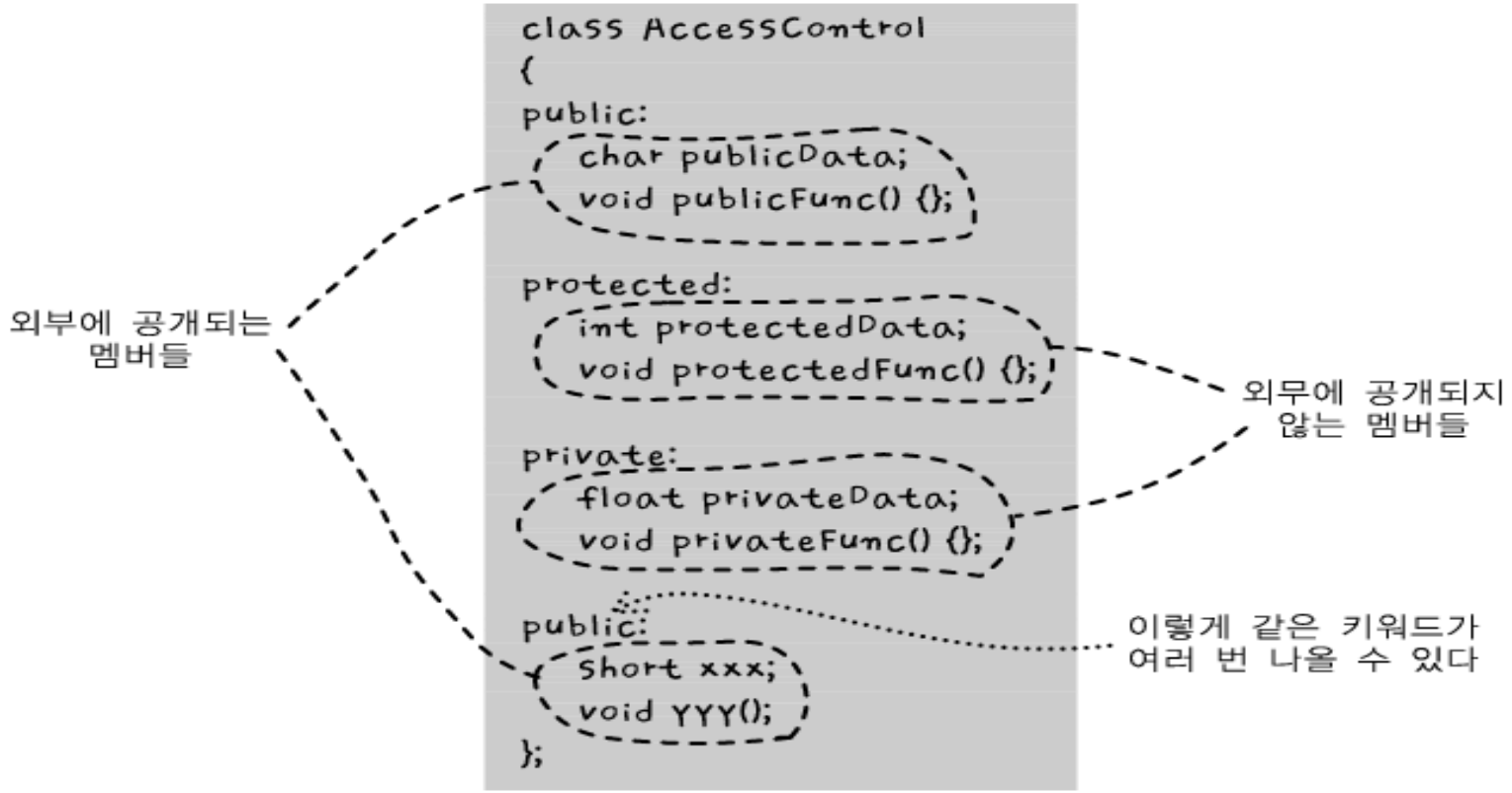
C++ 프로그래밍 기초

- 객체의 동적인 생성
  - 동적 메모리 할당을 사용해서 객체를 생성한 예

```
Point pt(50, 50);

Point* p1 = new Point();          // 디폴트 생성자 사용
Point* p2 = new Point(100, 100); // 매개변수가 있는 생성자 사용
Point* p3 = new Point(pt);        // 복사 생성자 사용

p1->Print();
p2->Print();
p3->Print();

delete p1;
delete p2;
delete p3;
```

C++ 프로그래밍 기초

- 접근 권한 설정하기
  - public : 외부에서의 접근 허용
  - protected, private : 외부에서 접근 금지



C++ 프로그래밍 기초

- 접근 권한 설정하기
  - 멤버의 접근 권한을 설정한 예

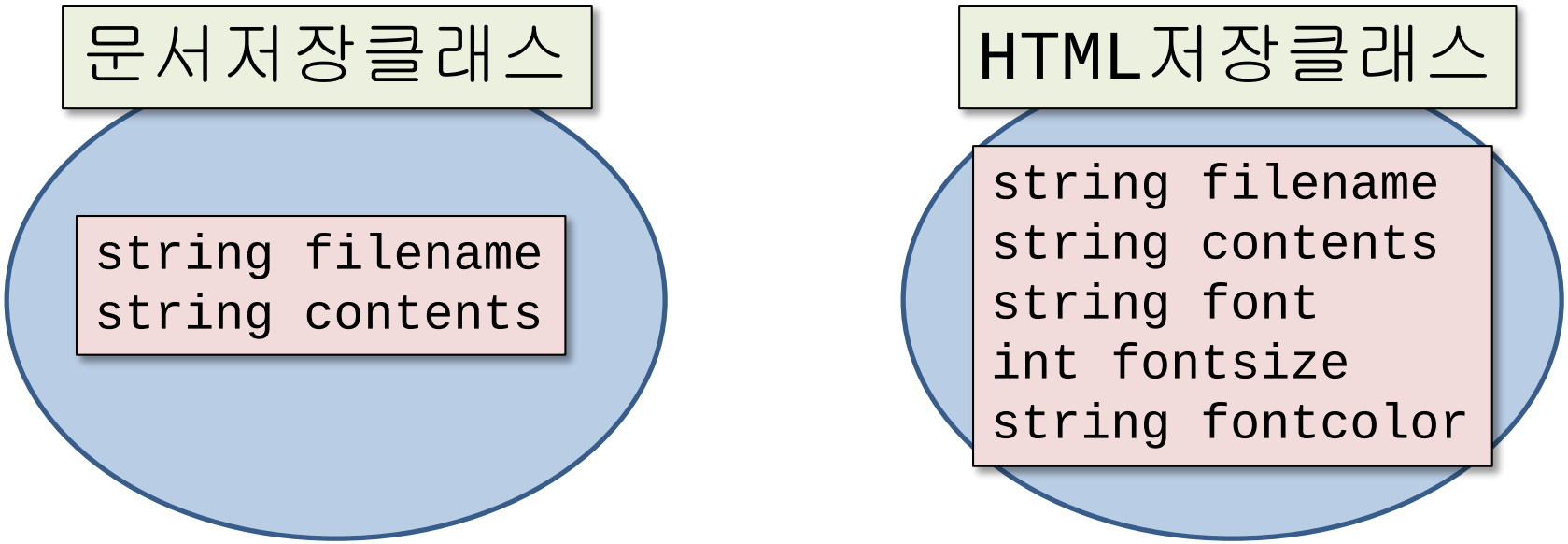
```
class AccessControl {
public:
            char publicData;
            void publicFunc() {};
protected:
            int protectedData;
            void protectedFunc() {};
private:
            float privateData;
            void privateFunc() {};
};

void main()
{
            // 객체를 생성하고, 각 멤버에 접근해보자
            AccessControl ac;

            ac.publicData = 'A';              // 성공
            ac.publicFunc();                  // 성공
            ac.protectedData = 100;           // 실패
            ac.protectedFunc();               // 실패
            ac.privateData = 4.5f;            // 실패
            ac.privateFunc();                 // 실패
}
```

C++ 프로그래밍 기초

- Inheritance(상속)과 containment (포함)
- Virtual function (가상 함수)
- Polymorphism (다형성)

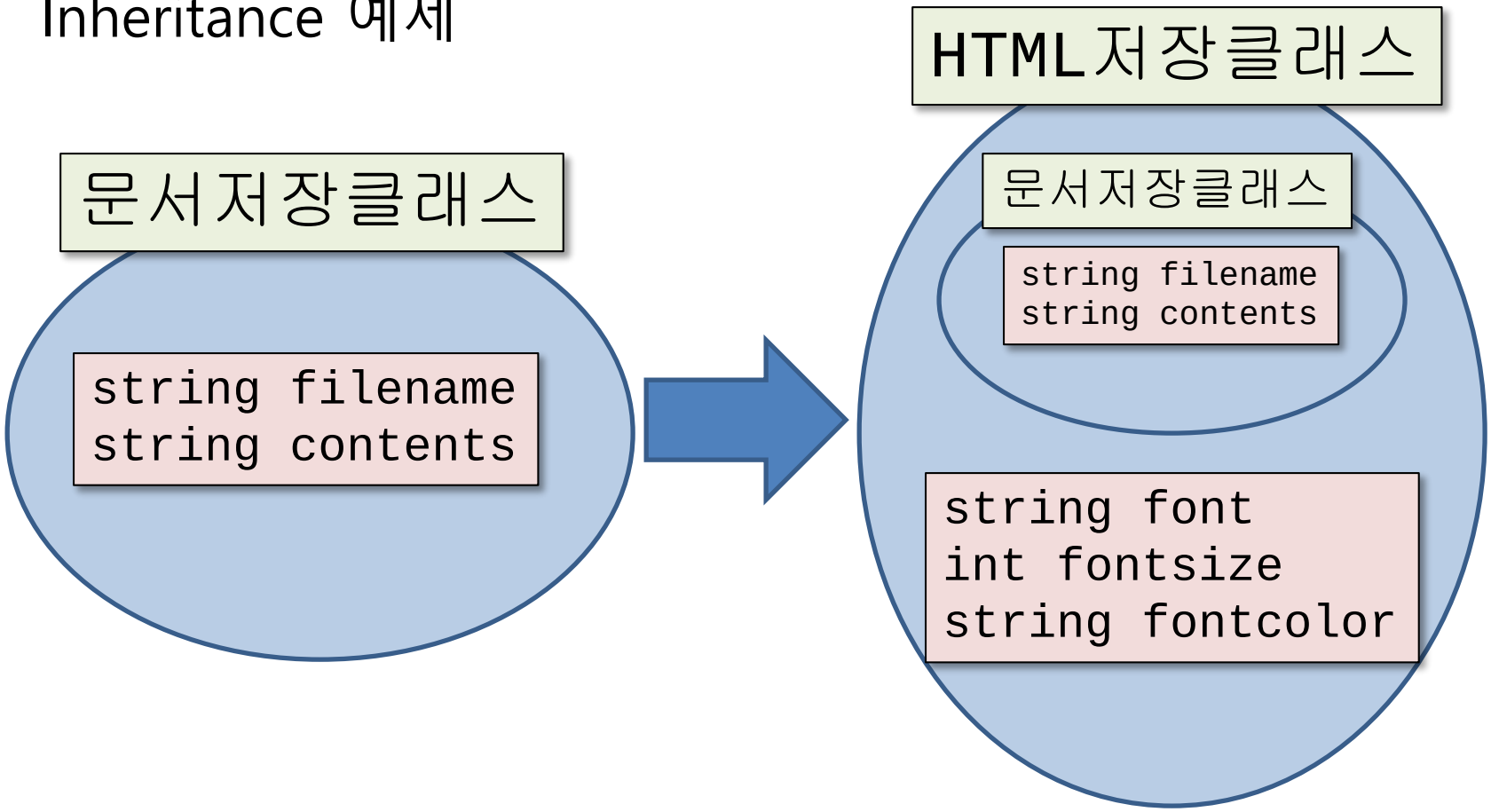
Inheritance

- Inheritance 예제



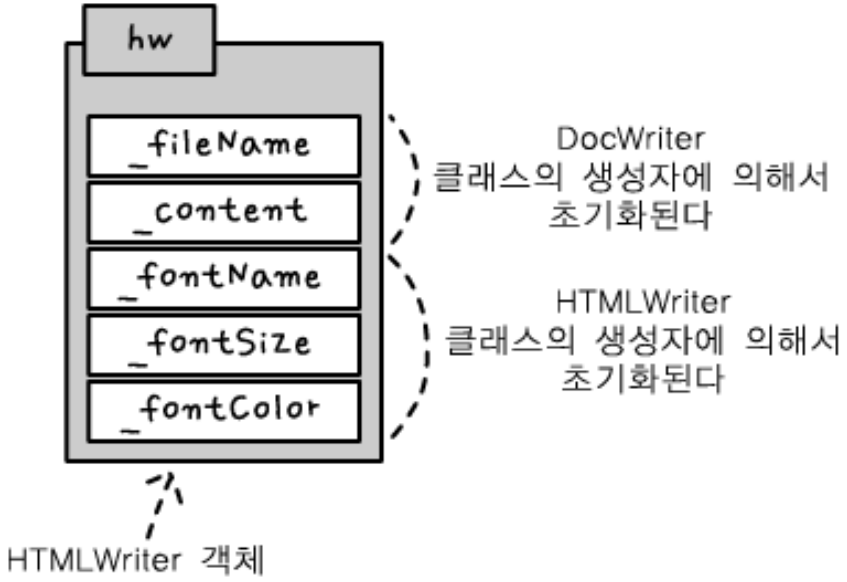
Inheritance

- Inheritance 예제



Inheritance

- Constructor & Destructor
  - Child object가 생성될 때, child class의 생성자 뿐만 아니라 parent class의 생성자도 호출됨



  - Parent class에 overloading된 여러 생성자가 있다면 그 중에서 어떤 생성자가 호출될 지 결정할 수 있음

Inheritance

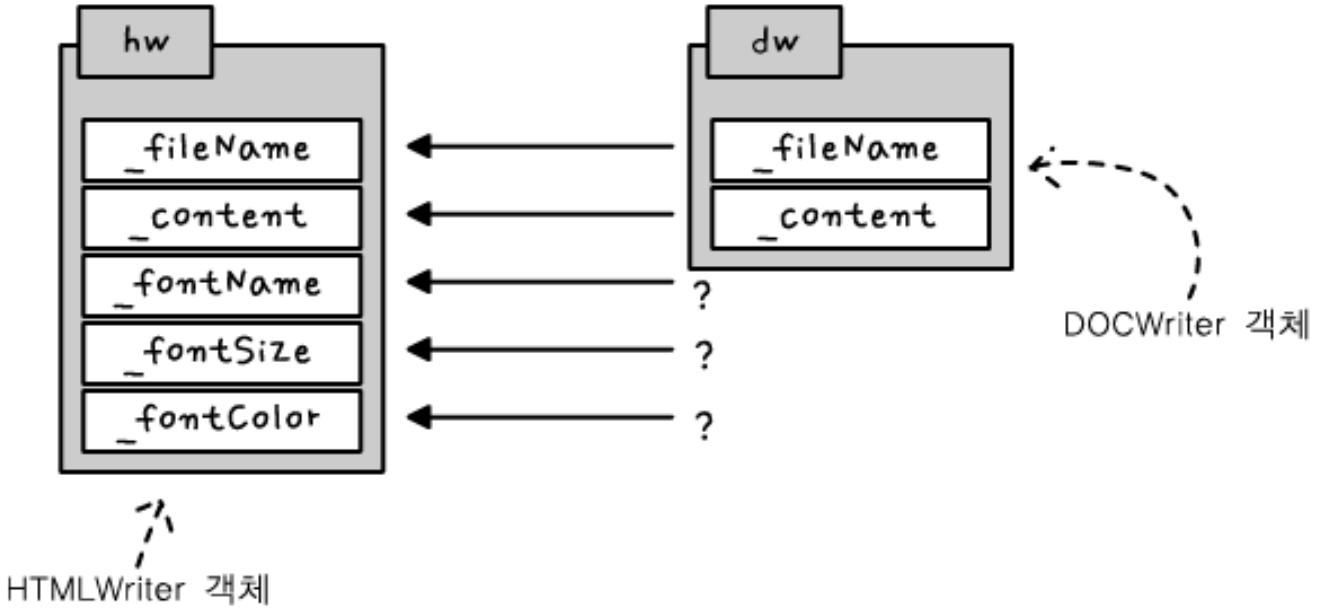
- Parent class의 생성자
  - Child class의 생성자에서 parent class의 생성자를 지정한 예

```
// HTMLWriter.h
class HTMLWriter : public DocWriter
{
public:
          HTMLWriter(void);
          HTMLWriter(const string& fileName, const string& content);
          ~HTMLWriter(void);
...

// HTMLWriter.cpp
HTMLWriter::HTMLWriter(const string& fileName, const string& content)
: DocWriter( fileName, content)       // 부모의 생성자 지정
{
          // 디폴트 폰트를 지정
          _fontName = "굴림";
          _fontSize = 3;
          _fontColor = "black";
}
...
```

Inheritance

- Parent와 child object 간의 대입 1
  - Parent class의 object를 child class의 object에 대입할 수 **없음**

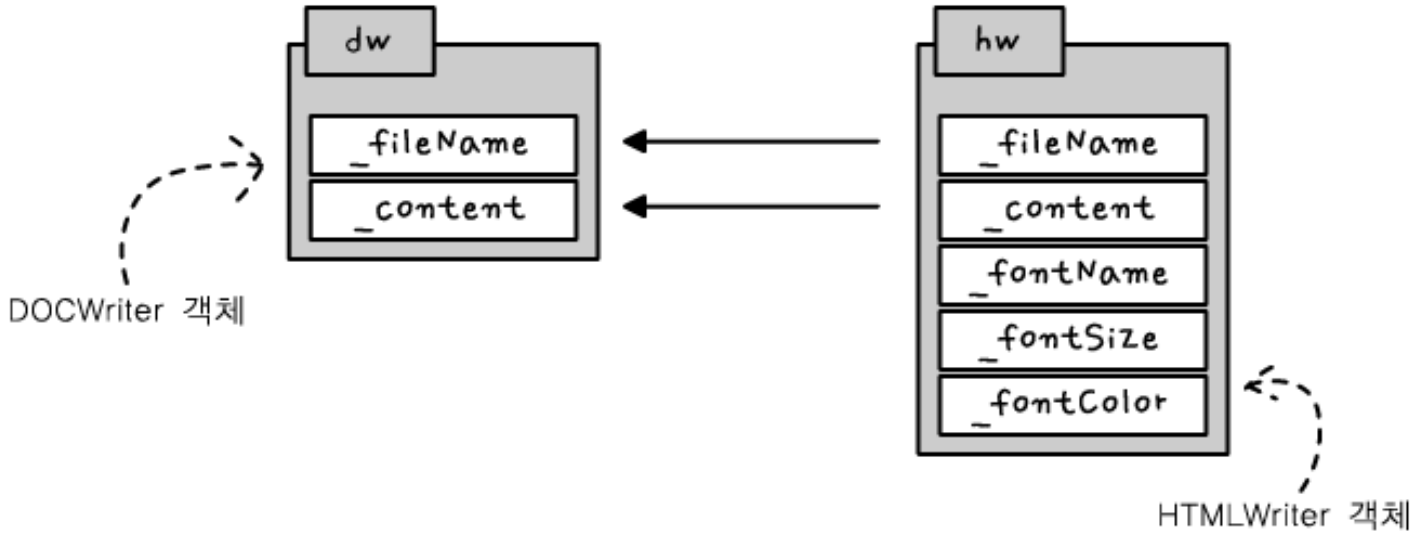
```
HTMLWriter hw;    // 자식 클래스의 객체 생성
DocWriter dw;     // 부모 클래스의 객체 생성

// 부모 클래스의 객체를 자식 클래스의 객체로 대입
hw = dw;          // Error!!
```



Inheritance

- Parent와 child object 간의 대입 2
  - Child class의 object를 parent class의 object에 대입할 수 **있음**

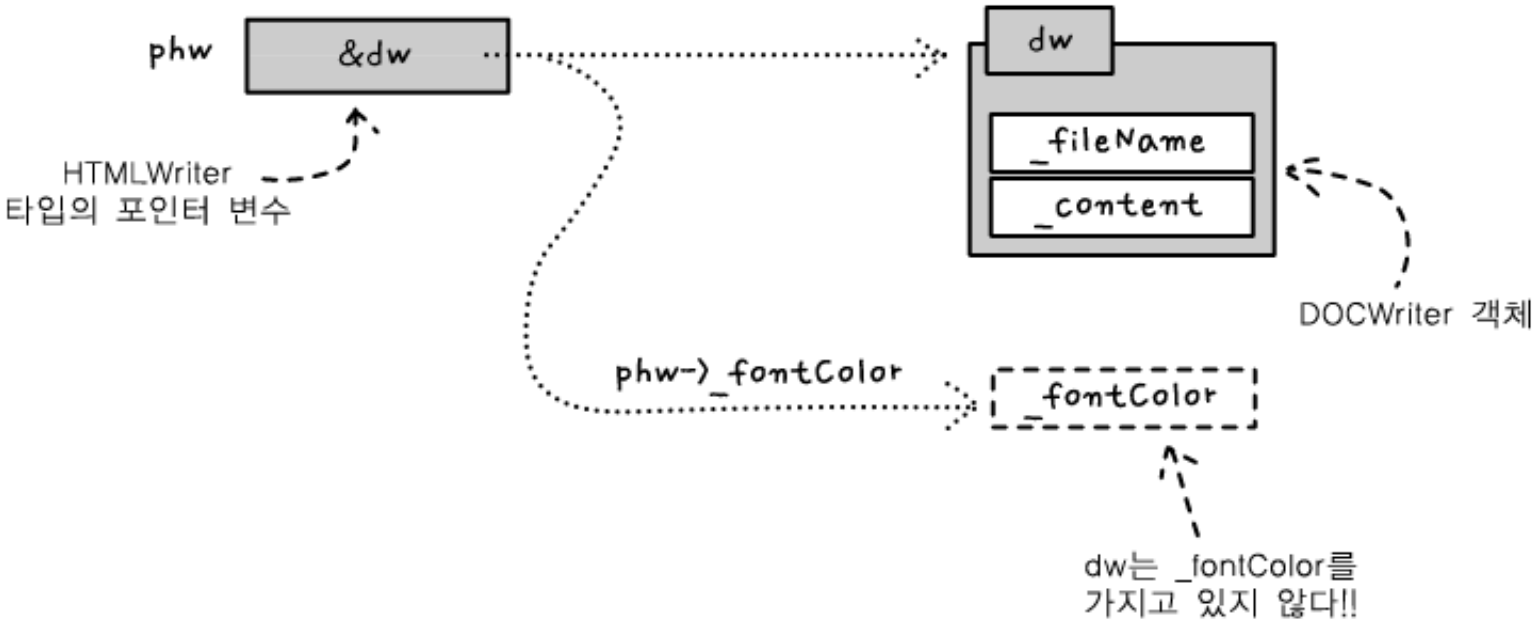
```
HTMLWriter hw;    // 자식 클래스의 객체 생성
DocWriter dw;     // 부모 클래스의 객체 생성

// 자식 클래스의 객체를 부모 클래스의 객체로 대입
dw = hw;          // OK
```



Inheritance

- Pointer의 type 변환 1
  - Child class의 pointer로 parent object를 가리킬 수 **없음**

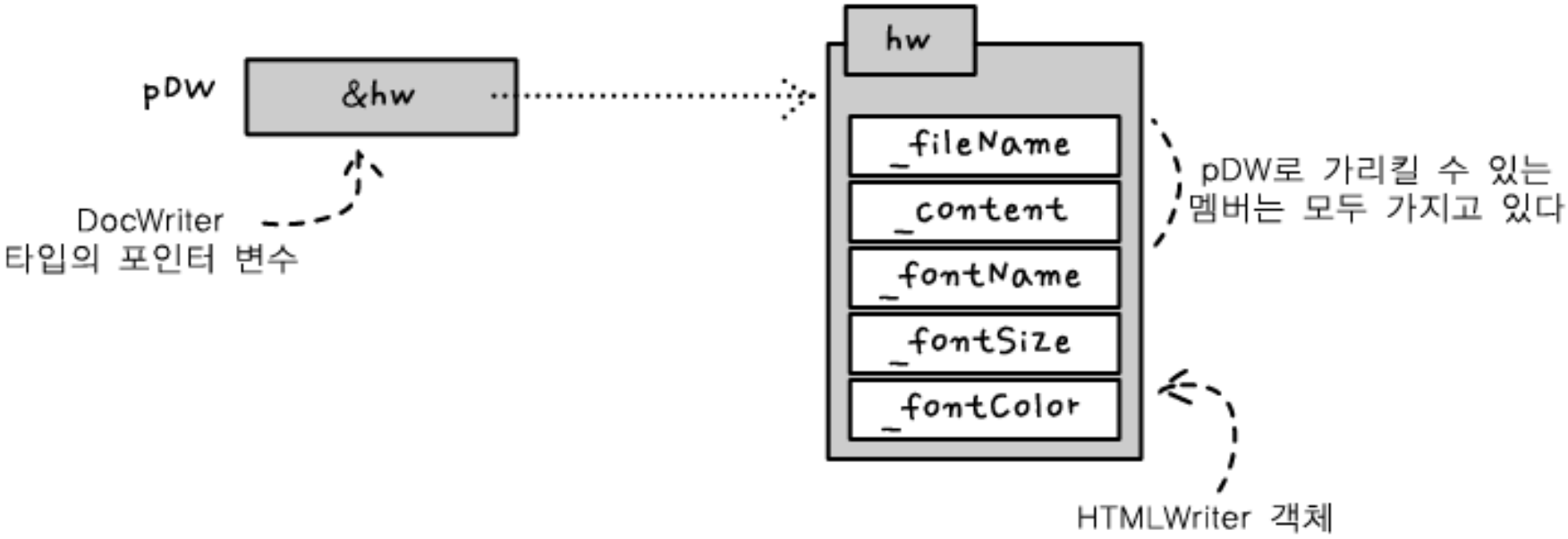
```
DocWriter dw;                  // 부모 클래스의 객체 생성

// 자식 클래스의 포인터 변수로 부모 객체를 가리킨다.
HTMLWriter* phw = &dw;     // Error!!
```



Inheritance

- Pointer의 type 변환 2
  - Parent class의 pointer로 child object를 가리킬 수 **있음**

```
HTMLWriter hw;              // 자식 클래스의 객체 생성

// 부모 클래스의 포인터 변수로 자식 객체를 가리킨다.
DocWriter* pDW = &hw;       // OK
```



Inheritance

- Reference의 type 변환
  - Reference의 경우도 pointer와 동일한 규칙
  - 자식 클래스의 레퍼런스로 부모 객체를 참조할 수 없음

```
DocWriter dw;              // 부모 클래스의 객체 생성

// 자식 클래스의 레퍼런스 변수로 부모 객체를 참조한다.
HTMLWriter& hw = dw;       // Error!!
```

  - 부모 클래스의 레퍼런스로 자식 객체를 참조할 수 있음

```
HTMLWriter hw;             // 자식 클래스의 객체 생성

// 부모 클래스의 레퍼런스 변수로 자식 객체를 참조한다.
DocWriter& dw = hw;        // OK
```

Inheritance

- Access control: 접근 제어 키워드

	자신의 멤버 함수에서 접근	자식 클래스의 멤버 함수에서 접근	외부에서 접근
private 멤버	O	X	X
protected 멤버	O	O	X
public 멤버	O	O	O

(O: 접근 가능, X: 접근 불가)

- Access control guideline
  - 외부로부터 숨겨야 하는 member는 protected로 지정
  - 그 밖의 경우는 public으로 지정
  - 반드시 자식 클래스에 숨기고 싶다면 private로 지정

Inheritance

- Access control: 상속과 관련해서 접근 제어 키워드 실험

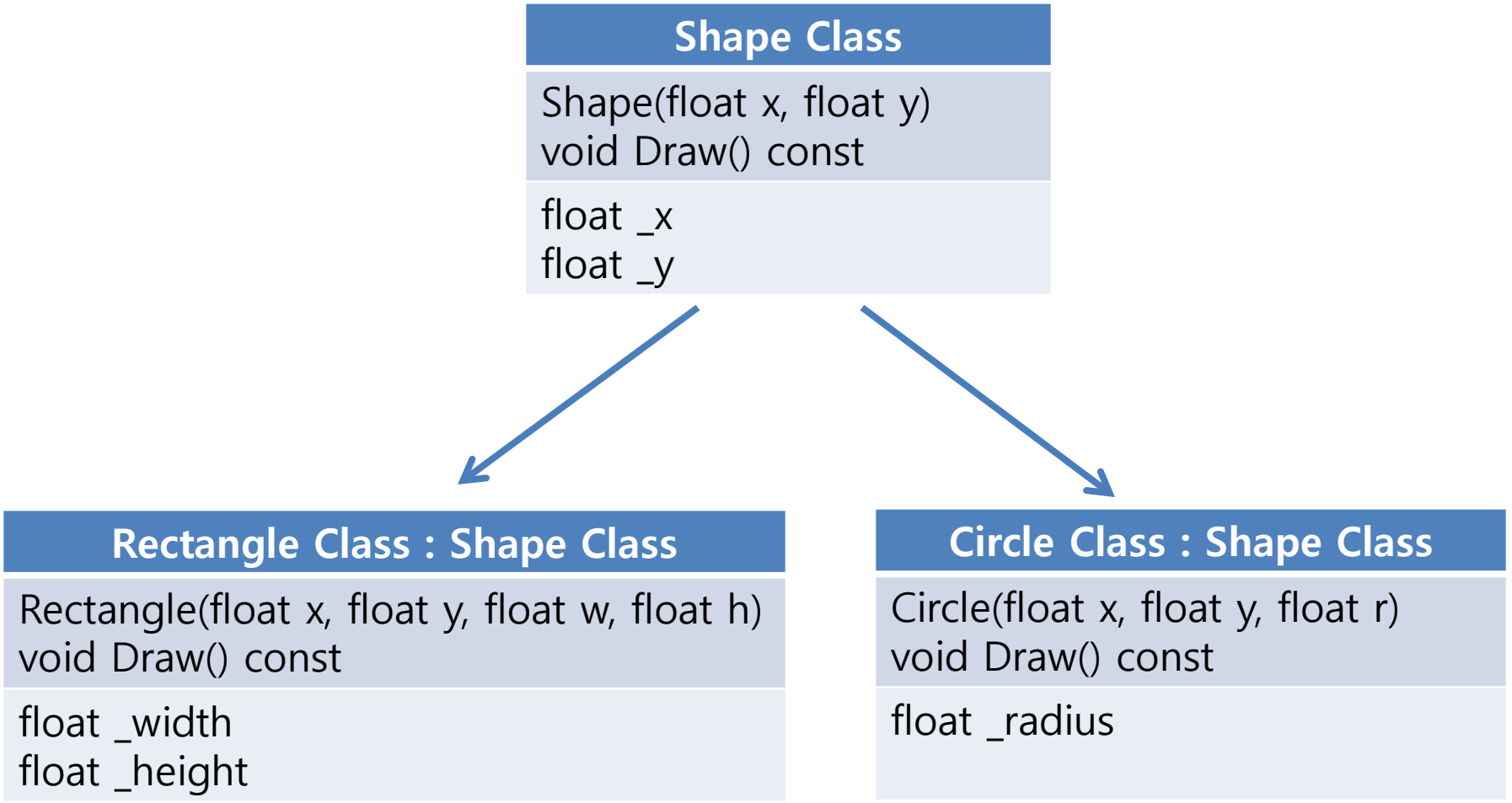
```
class Parent
{
private:
          int priv;
protected:
          int prot;
public:
          int pub;
};

class Child : public Parent
{
public:
          void AccessParents()
          {
                    int n;
                    // 부모의 멤버에 접근을 시도
                    n = priv;// 실패
                    n = prot;// 성공
                    n = pub; // 성공
          }
};
```

Inheritance

- Has-a 관계와 Is-a 관계
  - Has-a 관계 : A 가 B 를 가지고(포함하고) 있는 관계
    - 예) 자동차는 타이어를 가지고 있다.
  - Is-a 관계 : A 가 B 인 관계
    - 예) 사과는 과일이다.

- Containment(포함)과 inheritance(상속)을 구분해서 사용하기 위한 가이드라인
  - Has-a 관계의 경우에는 포함을 사용
  - Is-a 관계의 경우에는 상속을 사용

Practice: Inheritance



Practice: Inheritance

- Shape Class 정의
  - Member variable: **float _x,_y**
  - Constructor: **Shape(float x, float y)**
  - Member function: **void Draw() const**
    - 아래와 같은 내용 print
      [SHAPE] position = ('_x값', '_y값')

- Rectangle Class 정의
  - Shape Class로부터 상속
  - Member variable: **float _width, _height**
  - Constructor: **Rectangle(float x, float y, float w, float h)**
  - Member function: **void Draw() const**
    - 아래와 같은 내용 print
      [RECTANGLE] position = ('_x값', '_y값'), size = ('_width', '_height')

- Circle Class 정의
  - Shape Class로부터 상속
  - Member variable: **float _radius**
  - Constructor: **Circle(float x, float y, float radius)**
  - Member function: **void Draw() const**
    - 아래와 같은 내용 print
      [CIRCLE] position = ('_x값', '_y값'), radius = '_radius'

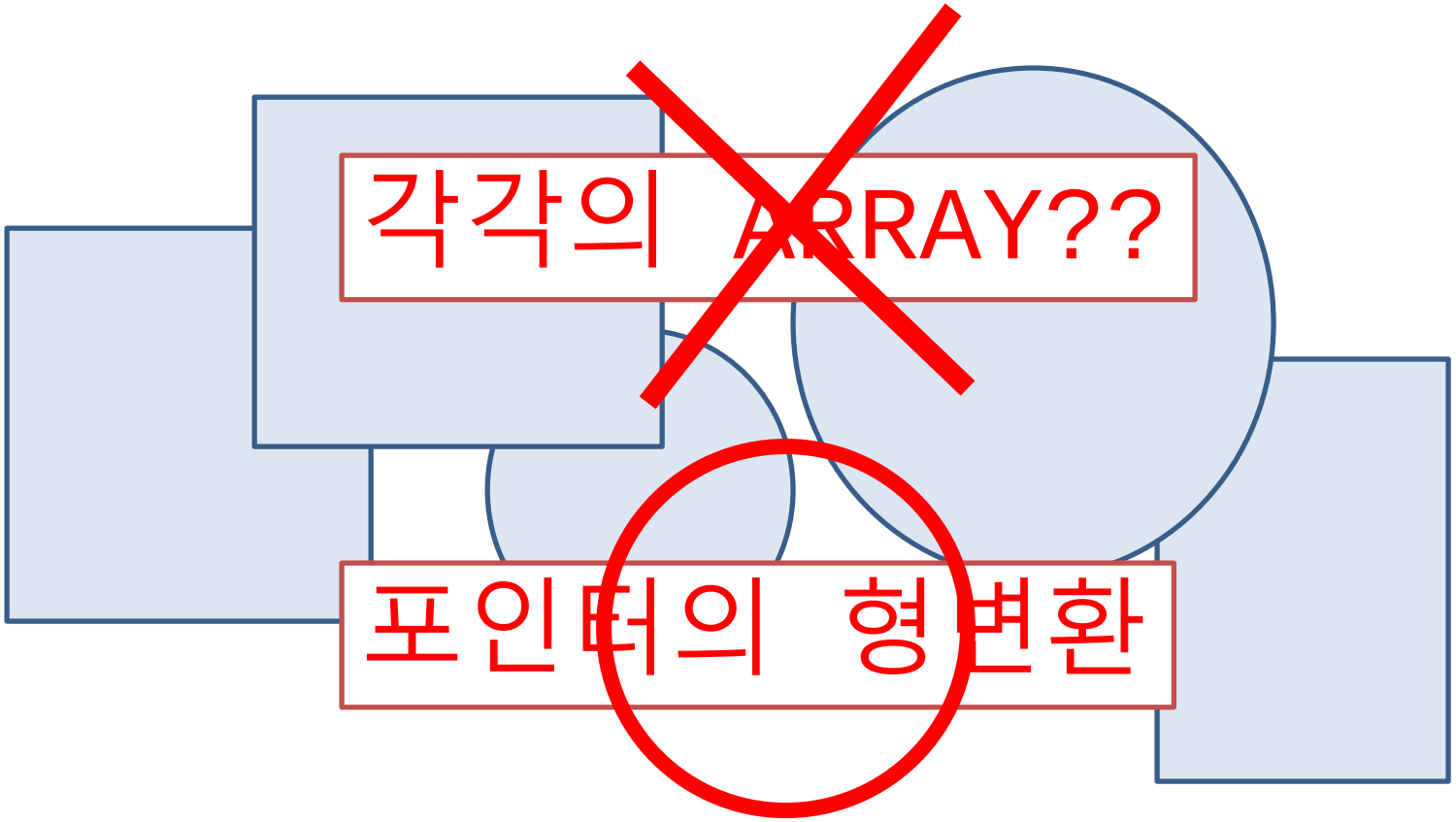
Practice: Inheritance

- Shape Class test

```
// main.cpp
int main()
{
        Shape a(100,40);
        Rectangle b(120,40,50,20);
        Circle c(200,100,50);

        a.Draw();
        b.Draw();
        c.Draw();

        return 0;
}
```

Virtual Function

- 여러 가지 형태의 도형을 저장하고 싶다



Virtual Function

- Pointer의 type 변환
  - Parent class의 pointer로 child object를 가리킬 수 **있음**

```
HTMLWriter hw;              // 자식 클래스의 객체 생성

// 부모 클래스의 포인터 변수로 자식 객체를 가리킨다.
DocWriter* pDW = &hw;       // OK
```

- Parent class의 pointer로 child object를 가리키기
  1. Parent(base) class pointer의 array을 설정
  2. 필요 시 마다 new를 이용 child class 생성
  3. 생성된 child class의 address를 parent class pointer에 할당
  4. 다 사용 했으면 delete를 이용 memory 해제

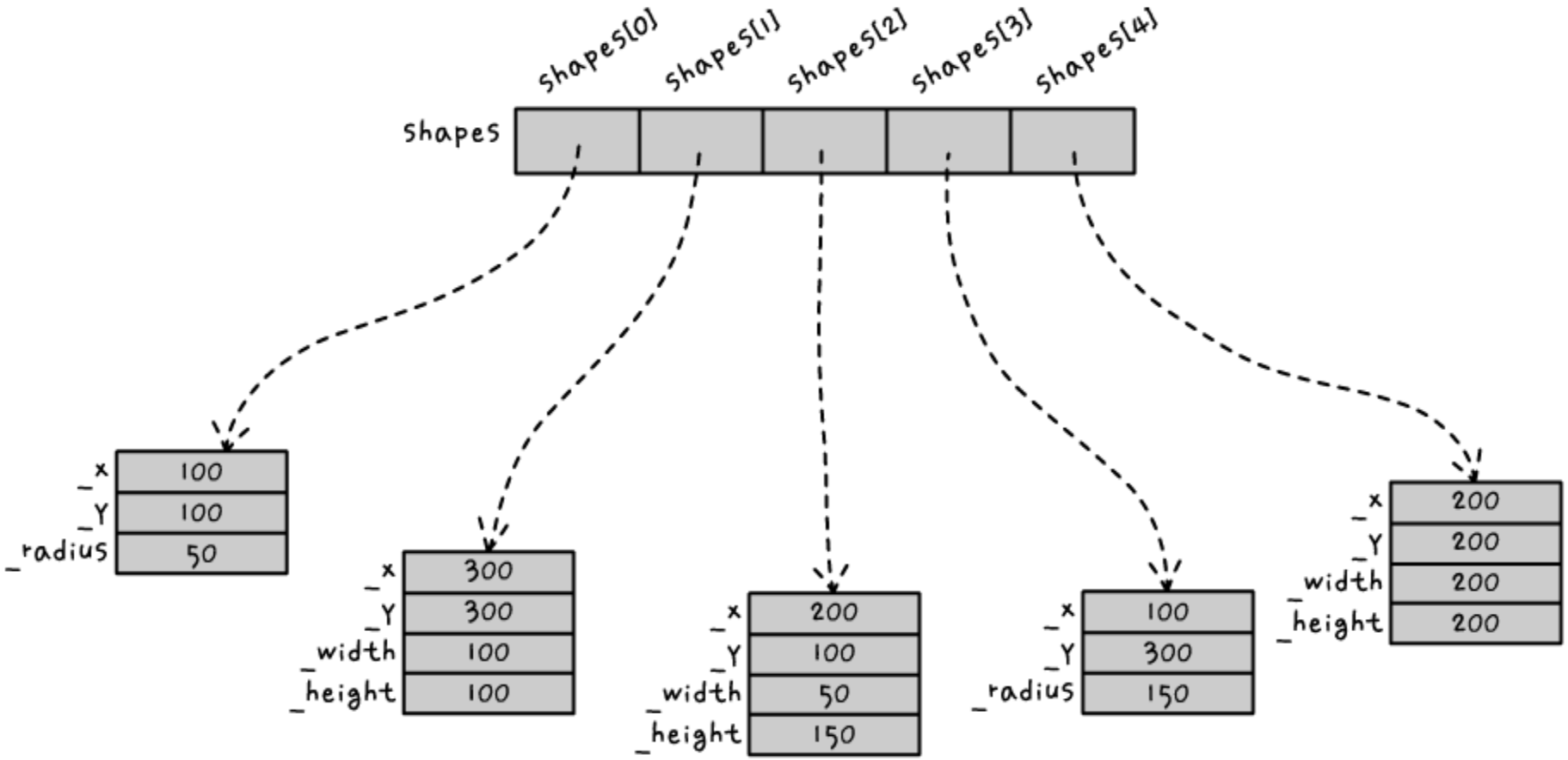
Practice: Virtual Function

- Shape class의 object들을 배열에 담아서 사용하는 예

```
// main.cpp
int main()
{
     Shape* shapes[5] = {NULL};

     shapes[0] = new Circle( 100, 100, 50);
     shapes[1] = new Rectangle( 300, 300, 100, 100);
     shapes[2] = new Rectangle( 200, 100, 50, 150);
     shapes[3] = new Circle(100, 300, 150);
     shapes[4] = new Rectangle( 200, 200, 200, 200);

     for (int i = 0; i < 5; ++i)
        shapes[i]->Draw();

     for (int i = 0; i < 5; ++i)
     {
        delete shapes[i];
        shapes[i] = NULL;
     }

     return 0;
}
```

Practice: Virtual Function

- Array와 object들의 메모리 구조



Practice: Virtual Function

- Draw의 실행 결과

```
[Shape] Position = ( 100, 100)
[Shape] Position = ( 300, 300)
[Shape] Position = ( 200, 100)
[Shape] Position = ( 100, 300)
[Shape] Position = ( 200, 200)
Press any key to continue
```

➔ Parent(base) class의 함수가 호출?

Virtual Function

- Parent class의 pointer로 child class를 가르킬 경우:
  - 같은 이름의 function의 실행은 **pointer type**이 우선

- Object의 실제 내용에 따라 불릴 순 없을까?
  - Function이 **compile**시에 결정되지 않고, **runtime**시 결정
  - Member function의 **동적인 선택(dynamic binding)**
  - Polymorphism(다형성): 객체가 자신의 본래 정보와 동적으로 연결되는 것

➔ **Virtual Function**

Virtual Function

- Parent class에서 function **선언**부분에 virtual 이라고 추가
  - Virtual function도 **상속**되기 때문에 child에게도 붙일 필요 없음
- Draw 문제의 해결
  - Shape class의 Draw() 선언부분 수정한 예

```
// Shape.h
class Shape
{
	virtual void Draw() const;
};

// Rectangle.h
class Rectangle : public Shape
{
	void Draw() const;
};
```

```
[Circle] Position = ( 100, 100) Radius = 50
[Rectangle] Position = ( 300, 300) Size = ( 100, 100)
[Rectangle] Position = ( 200, 100) Size = ( 50, 150)
[Circle] Position = ( 100, 300) Radius = 150
[Rectangle] Position = ( 200, 200) Size = ( 200, 200)
Press any key to continue
```

Polymorphism

- 다른 분야에서의 polymorphism(다형성)의 의미

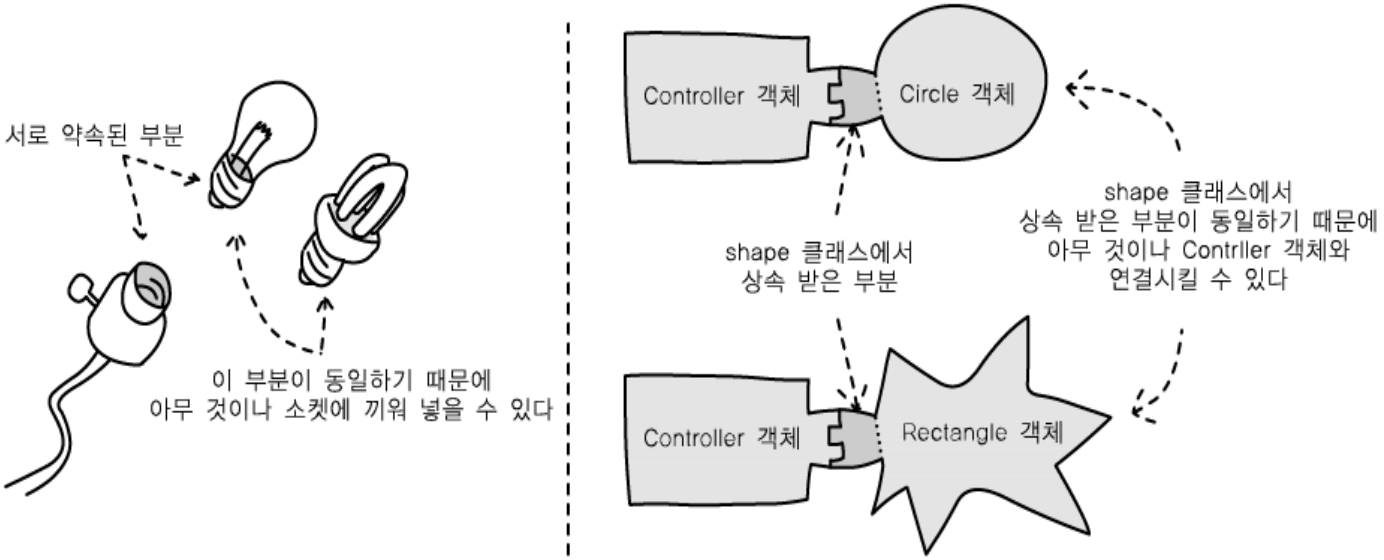
생물학	동일종의 생물이면서 형태나 성질이 다르게 보이는 다양성. 생물은 본래 동일종이라도 완전히 일치하는 개체는 거의 없으므로, 이 말은 상대적으로 현저한 차이가 있을 경우에 한해서 사용한다. 다만, 암수의 성별에 따른 2차 성징(二次性徵)의 차이에는 쓰지 않는다.
화학	화학조성이 같은 물질로써 결정구조를 달리하는 것. 다형(多形), 동질이형(同質異形), 동질다상(同質多像), 동질이정(同質異晶)이라고도 한다. 동질다상은 결정구조의 수에 따라 동질이상(同質二像), 동질삼상(同質三像) 등으로 나뉜다.

- OOP에서의 polymorphism
  - Type에 관계 없이 동일한 방법으로 다룰 수 있는 능력
  - 예) Circle이나 Rectangle object들을 type에 상관 없이 Shape object처럼 다룰 수 있음

Polymorphism

- Polymorphism은 object 간의 coupling(결합)을 약하게 만들어서, 객체 간의 연결을 유연하게 해줌

```
// 도형을 원점으로 이동하는 함수
void Controller::MoveToOrigin(Shape *p)
{
        p->Move( 0, 0 );
        p->Draw();
}
```



Polymorphism

- 다양한 형태의 member function
  - Inheritance와 관련된 member function의 종류
    - 일반적인 member function
    - Virtual function

- 어떤 종류의 member function을 만들까?
  - 기본형태 → member function
  - Polymorphism → virtual function

Polymorphism

- Overloading
  - 호출 인자가 다를 경우에 알맞은 것을 선택
  - 예)
    ```
    void DoIt( );
    void DoIt(int i);
    ```

- Overriding
  - Parent object의 function을 child object에서 재정의
  - 예)
    ```
    parent::DoIt( );
    child::DoIt( );
    ```

Q & A